Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 207–217
www.elsevier.com/locate/entcs

Dynamic Software Assembly for Automatic Deployment-oriented Adaptation

Anthony Savidis[1]

Institute of Computer Science
Foundation for Research and Technology - Hellas (FORTH)
Heraklion, Crete, Greece

Abstract

The notion of software adaptation considered in this paper relates to the capability of making software systems adjustable to varying deployment requirements. In this context we seek for the necessary runtime infrastructure to allow software systems adapt on the fly to the particular execution requirements. The primary assumption is that the constituent components of a software system may have to be provided with alternative incarnations, each potentially addressing varying deployment needs. In this context, adaptation is treated as a runtime function of the system itself, realising a component and assembly process, since the deployment-specific parameters are only known upon execution start-up.

Keywords: software adaptability, dynamic software assembly, deployment-oriented adaptation.

1 Introduction

The need for software adaptability has been identified in [1], mainly emphasizing static software properties such as extensibility, flexibility and performance tunability, without negotiating the automatic and dynamic software assembly. Similarly, in [2], adaptability is also considered a key static property of software components, which can be pursued through aspectual decomposition, i.e., by employing aspect-oriented programming methods. In this paper, we are targeted in the engineering of software systems capable to dynamically activate alternative implementation versions of embedded software components

[1] Email: as@ics.forth.gr

1571-0661 © 2005 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.08.045

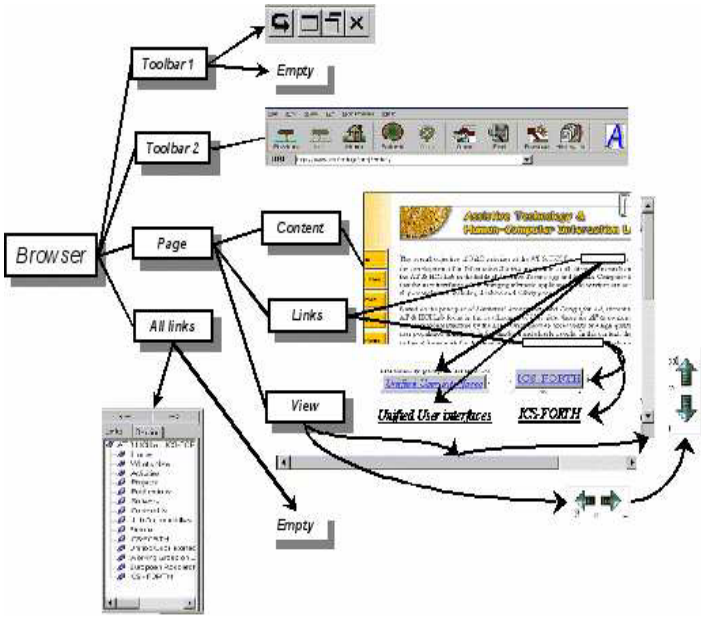

Fig. 1. The hierarchical User Interface component structure of an adaptable browser; arrows indicate alternative implementations of components (empty indicates a component can be entirely omitted).

through a runtime decision process, which relies on deployment-oriented decision parameters. To provide a more precise idea regarding dynamic software assembly based on deployment requirements, the application of the reported work in the context of dynamic User Interface assembly will be supplied. In this context, deployment parameters concerned individual user profiles, including abilities, preferences, expertise, etc. The dynamically assembled software artifacts concerned User Interface components.

In Figure 1, an excerpt from the Use Interface component structure of the AVANTI web browser [3] is shown; arrows indicate interface components whose activation and graphical embedding takes place at start-up conditionally, depending on the individual user profile (the deployment parameters for User Interfaces were actually user profile parameters). The adaptation-oriented decision logic for the cases of Figure 1 where alternative implementations exist is provided in Figure 2.

Following Figure 2, the decision logic engages user attributes (variables) within if-then-else rules that encompass activation statements. For example, "Links" is a component family with two alternative implementations, each associated uniquely with a descriptive identifier, e.g. "LinksAsButtons" and "LinksAsUnderlinedText". This implies that the container of the "Links" fam-

Component identifier	*Decision logic*

"Toolbar 1"	`if (user."CanUseUpperLimbs" == false) then`
	`  activate "ScanningToolbar";`
	`else`
	`  activate "Empty";`

"Links"	`if (user."WebExpertise" in {"Naïve", "Casual"}) then`
	`  activate "LinksAsButtons";`
	`else`
	`  activate "LinksAsUnderlinedText";`

"View"	`if (user."WebExpertise" in {"Naïve", "Casual"}) then`
	`  activate "EmbeddedScrollbars";`
	`else`
	`  activate "ScrollWindow";`

"All links"	`if (user."WebExpertise" in {"Naïve", "Casual"}) then`
	`  activate "LinkEnumeration";`
	`else`
	`  activate "Empty";`

Fig. 2. The decision logic engaging user profile parameters (deployment profile) for adapted User Interface component selection and activation

ily is capable to physically embed either of the alternatives, while the choice as to which "Links" instantiation is to be activated is taken by executing the decision logic upon application start-up. In this context, because of the fact that the "Links" component may have multiple alternative realisations it is called a polymorphic component, meaning it can be met in different executions of the same interactive application with different forms. The software engineering approach for dynamic User Interface adaptation according to user profiles is extensively described in [4]. In this paper, the generalisation of dynamic User Interface assembly is reported, targeted in the implementation of software systems with the following properties: (a) they encapsulate alternative component implementations reflecting varying deployment requirements; (b) they are architecturally organized in ways enabling alternative implementations of polymorphic architectural components to be easily accommodated; (c) they encapsulate decision making driven by deployment parameter values; and (d) they perform a runtime software assembly process bringing together the necessary constituent components that bets-fit the particular deployment profile.

2 Architectural polymorphic decomposition

The key architectural implication due to the functional requirement for dynamic adaptation-oriented software assembly is the need for organization of implemented software components so as to enable dynamic architectural containment hierarchies. It should be noted that since containment concerns architectural decomposition relationships, i.e. components that logically encompass other components, containment always reflects a hierarchical struc-

ture. Other architectural views also exist, like dependency (or call) graphs, data exchange, etc., but those are not employed for the software assembly problem in our context. Overall, every software system has a hierarchical architectural view of its constituent components, which is actually of key importance when targeted in dynamic software assembly from runtime selected constituent components.
In non-adaptable monomorphic software applications, developers typically program the hierarchical (containment) structure of architectural components through hard-coded associations that are determined during development time. However, in the context of adapted software delivery, the component containment hierarchies should support two key features: (a) parent-child associations are always decided and applied during runtime; and (b) multiple alternatively candidate contained-instances are expected for composite components. The component organization method of dynamic polymorphic containment hierarchies is illustrated in the Figure 3. Following Figure 3, PL indicates the polymorphism factor, which provides the total number of all potential different run-time incarnations of a software component, recursively defined as the product of the polymorphic factors of constituent component classes. Practically, the actual number of plausible distinct software versions is less than PL, while it can be extracted by analysing the "diversity" of the deployment parameters. But since the deployment requirements may differentiate even per a component basis, the PL number does not only serve as a theoretical upper bound.

The hierarchical User Interface component structure of an adaptable browser; arrows indicate alternative implementations of components (empty indicates a component can be entirely omitted).

3 Dynamic assembly process

Since the hierarchical component-containment structure engages components, which can have alternative incarnations, it is implied that either the contained or the container components may vary. As a result, this hierarchical structure is not monomorphic, but reflects also a polymorphic discipline. In this context, the dynamic assembly process reflects the hierarchical traversal in the polymorphic containment hierarchy (see Figure 4), starting from the root component, to decide, locate, instantiate and initiate appropriately every target contained component. This process primarily concerns the architectural components that are actually polymorphic, i.e. architectural container components designed with alternative deployment-oriented decompositions.

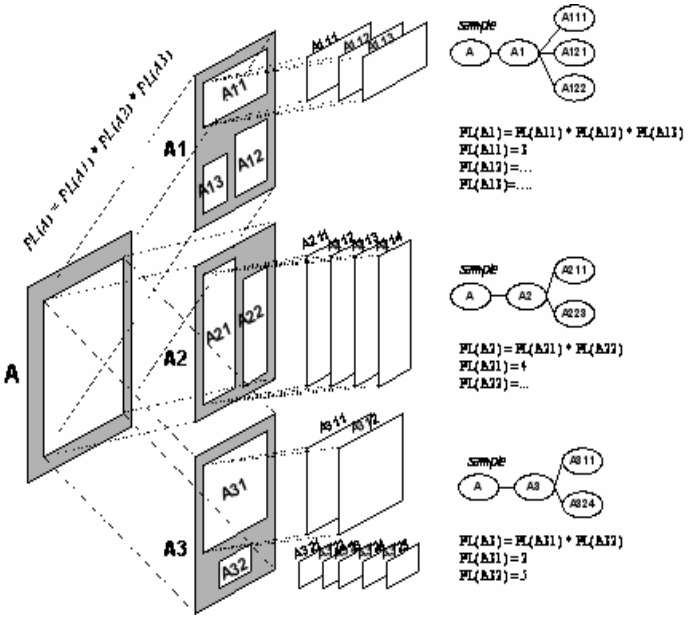

Fig. 3. Illustration of polymorphic architectural containment for software components, showing the potential for multiple implementation instantiations of embedded components; it should be noted that the hierarchical architectural containment view is only of importance for dynamic software assembly

From the implementation point of view, the following software design decisions can be made:

- The containment-oriented architecture-component hierarchy has been implemented as a tree data structure, with polymorphic nodes triggering decision making sessions;
- Software components have been implemented as distinct independent software modules, implementing architecture-role generic Application Programming Interfaces (APIs), while exposing a singleton control-API for dynamic instantiation and name-based lookup;
- The software assembly procedure is actually carried out via two successive hierarchical passes:
  - Execution of decision sessions, to identify the specific selections for polymorphic architectural contexts, that will be part of the eventually delivered software;
  - Software assembly and start-up, through instantiation and initiation of all decided components.

Although the previous process is only conceptually illustrated under Fig-

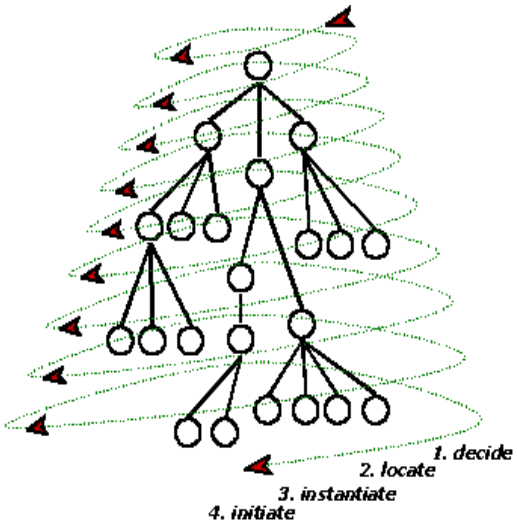

Fig. 4. The traversal of the hierarchical containment architectural structure to: decide, locate, instantiate and initiate software components

ure 4, its implementation is quite straightforward when concerning component that are singleton classes: each alternative singleton derives from the basic base-class API (component-specific), while all singleton pointers are populated in a hash-map; the initial selection is simply made via name-based look-up. The implementation of dynamic assembly becomes much harder when polymorphic components concern normal program classes, instances of which are made explicitly via statements within the program source-code. More specifically:

- Let *A* be a programmer-defined class.
- *Aa;* and *new A();* are two example statements for explicit instantiation of class A in the program source code.
- Let *A1,,An* be alternative deployment-oriented implementations of *A;* we need to allow instantiations to concern *Ai*, assuming *Ai* implementation is chosen upon decision making.
- We want to provide a generic instantiation style of the form: *A::Construct ("Ai")*, thus supporting parameterization of the specific class name.

The previous required style signifies a departure from the traditional style of hard-coded class instantiations in the program source code to parameterized instantiations, enabling the class identifier to be supplied as a string argument. In other words, instead of new Ai, we want to support *A::Construct (A::GetDecidedClassId())*, where *GetDecidedClassId* is a static function re-

turning the decided *A* version from *A1,,An*. This represents a radically different perspective in class deployment, enabling orthogonal expansion of mutually exclusive class versions, while emphasizing deployment according to the base class API. The implementation of this technique is illustrated in the source code pattern of Figure 5. As shown in Figure 5, every class version implements a constructor functor class that is registered upon static class initialization in the class-specific dispatch table; this functor class, named Constructor, is responsible for dynamic class instantiation by calling the appropriate overloaded class constructor. This technique is a variation of double / dual dispatching. At the bottom of Figure 5, the parameterized deployment style is shown, with the traditional style of hard-coded class use put in comments. Clearly, the new style requires a little more code typing, however, it emphasizes far better deployment based on the basic class API, i.e. *Base**, while completely hiding the different class versions.

Additionally, it supports orthogonal extension of class versions, since the implementation of the dispatching method within the Base class is not dependent on derived classes; hence, once new derived classes are implemented according to the suggested pattern, those become automatically engaged in the adaptation process. This technique is easier to implement once classes become available over a component-ware technology, as they are already delivered over proxy APIs. Also, in cases of languages enabling dynamic class loading, like Java or Action Script, dynamic loading of class versions is straightforward.

4 Key architectural ingredients

As it has been previously mentioned, the adopted notion of software adaptability reflects the functional properties of automatic software assembly, through decision-making that relies upon runtime software adaptation parameters. It should be noted that this is a fundamentally different target from formal methods related to software evolution, which focus on the automated transformation and evolution of software structures at development-time, according to diverse software requirements. The key architectural elements towards dynamic software assembly are:

- Hierarchical architectural view (component containment)
- Architectural context (sub-architecture that is subject to adaptation)
- Software component
- Software deployment parameters
- Software deployment scenarios
- Polymorphic architectural components

```
class Base {
    public:
       struct CtorArgs_string {
              std::string arg;
              CtorArgs_string (const std::string& s) : arg(s){}
       };
       struct CtorArgs_int {
              int arg;
              CtorArgs_int (int i) : arg(i){}
       };
       struct CtorArgs_void  { CtorArgs_void (void){}        };

       class Ctor {
              public:
              virtual Base* operator()(CtorArgs_string&) = 0;
              virtual Base* operator()(CtorArgs_int&) = 0;
              virtual Base* operator()(CtorArgs_void&) = 0;
       };
       static const std::string GetDecidedClassId (void);

       template <class Args> static Base* Construct (Args& args) {
              std::map<std::string, Ctor*>::iterator i;
              i = ctorMap.find(GetDecidedClassId());
              assert(i != ctorMap.end());
              return (*i->second)(args);
       }
    protected:
       static std::map<std::string, Ctor*> ctorMap;
};

class Derived : public Base {
       class Constructor;
       friend class Constructor;
       Derived (const std::string&);
       Derived (int);
       Derived (void);
    public:
       class Constructor : public Base::Ctor {
              public:
              Base* operator()(CtorArgs_string& a)
                     { return new Derived(a.arg); }
              Base* operator()(CtorArgs_int& a)
                     { return new Derived(a.arg); }
              Base* operator()(CtorArgs_void& a)
                     { return new Derived; }
       };
       static void Initialise (void)
              { ctorMap["Derived"] = new Constructor; }
};

Base* b = Base::Construct(Base::CtorArgs_void());
// Derived* d = new Derived;
```

Fig. 5. Implementing virtual destructors through runtime dispatching of class instantiations relying on class and argument type dispatching

- Alternative encapsulated components
- Architectural decomposition
- Architectural role component indexing

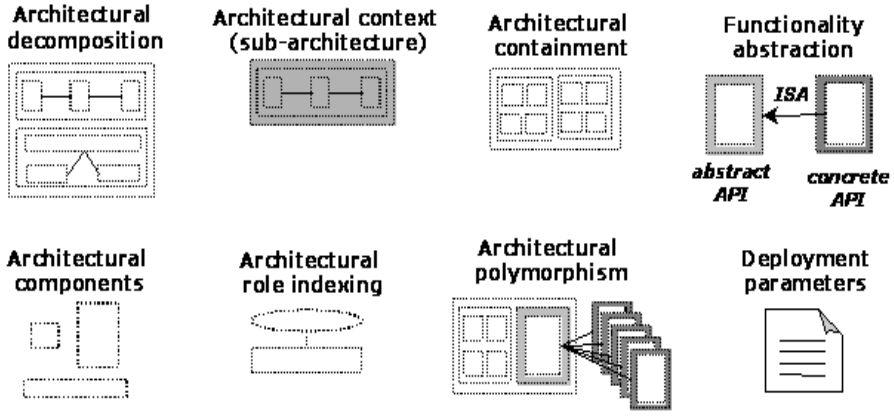

Fig. 6. The key architecture meta-elements for dynamic software assembly, emphasizing the capability to accommodate alternative mutually exclusive implementation of components and classes within the same architecture

- Architectural containment
- Functional-role abstraction APIs
- Mutually exclusive class versions

Those lead to an augmented vocabulary for the software architecture domain, mainly introducing the meta-elements necessary to accommodate runtime software assembly driven by decision-making for deployment adaptation, as illustrated in Figure 6.

5 Dynamic content delivery

In the context of the PALIO project [5], the software engineering method for dynamic software assembly has been effectively employed for adaptable information delivery over mobile devices to tourist users. The decision-making process was based on parameters such as nationality, age, location, interests or hobbies, time of day, visit history, and group information (i.e. family, friends, couple, colleagues, etc.). The information model reflected a typical relational database structure, while content retrieval was carried out using SQL queries in XML. In this context, in order to enable adapted information delivery, instead of implementing hard-coded SQL queries, query patterns have been designed, with specific polymorphic placeholders filled in by dynamically decided concrete sub-query patterns. For instance, as shown in Figure 7, particular data categories or even query operations may be left "open", with multiple alternatives, depending on runtime content-adaptation decision making.

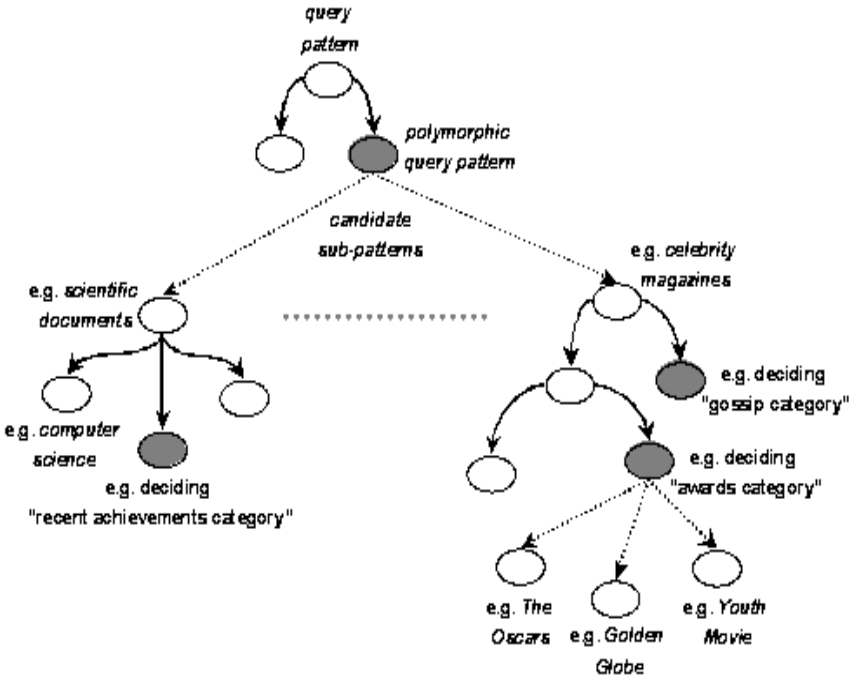

Fig. 7. Polymorphic query patterns for adaptable query formulation

6 Conclusions

This paper negotiates the software engineering of systems capable to realize a dynamic assembly behavior, from a pool of fully implemented software components, according to decision-making that is based on deployment-oriented requirements (in contrast to design-time decisions). During execution, the system reflects a runtime transformation behavior, in the sense that is capable to set-up itself on the fly according to the particular deployment requirements. To accomplish this behavior with design-time transformations, all plausible system versions, as combinations of the desirable components, need to be produced and delivered together. Clearly, this is an impractical method, while it does not allow the system to dynamically extend, e.g. by enabling downloading and installing new component versions addressing additional deployment needs.

References

[1] Fayad, M., Cline, M., *Aspects of Software Adaptability*, CACM Journal. **39(10)** (1996), 58-59.

[2] Netinant, P., C. A. Constantinides, T. Elrad, M. E. Fayad., *Supporting Aspectual Decomposition in the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks*, Proceedings of 3rd Workshop on Object-Orientation and Operating Systems ECOOP-OOOWS, Sophia Antipolis, France, (2000), 36-46.

[3] Stephanidis, C., Paramythis, A., Sfyrakis, M., Savidis, A. *A Case Study in Unified User Interface Development: The AVANTI Web Browser*. In User Interfaces for All, Stephanidis, C. (Ed), Lawrence Erlbaum, NJ, (2001), 525-568.

[4] Stephanidis, C., Savidis, A., Akoumianakis, D. *Universally accessible UIs: The Unified User Interface development*. Tutorial in the ACM Conference on Human Factors in Computing Systems (CHI 2001), Seatle, Washington, 31 March - 5 April. (2001), available at: http://www.ics.forth.gr/proj/at-hci/files/CHI_tutorial.pdf.

[5] Stephanidis, C., Paramythis, A., Zarikas, V., Savidis, A. *The PALIO Framework for Adaptive Information Services*. In A. Seffah, H. Javahery (Eds.), **Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces**. Chichester, UK: John Wiley and Sons Ltd, (2004), 69-92.